Evol. Intel. (2010) 3:171–186
DOI 10.1007/s12065-010-0042-z

RESEARCH PAPER

Composition of web services through genetic programming

Pablo Rodríguez-Mier · Manuel Mucientes · Manuel Lama · Miguel I. Couto

Received: 22 February 2010 / Revised: 11 August 2010 / Accepted: 23 September 2010 / Published online: 15 October 2010
© Springer-Verlag 2010

Abstract Web Services are interfaces that describe a collection of operations that are network-accessible through standardized web protocols. When a required operation is not found, several services can be compounded to get a composite service that performs the desired task. To find this composite service a search process in a, generally, huge search space must be performed. The algorithm that composes the services must select the adequate atomic processes and, also, must choose the correct way to combine them using the different available control structures. In this paper a genetic programming algorithm for web services composition is presented. The algorithm has a context-free grammar to generate the valid structures of the composite services and, also, it includes a method to update the attributes of each node. Moreover, the proposal tries to minimize the number of services, and looks for compositions with the minimum execution path. A full experimental validation with four different repositories with up to 1,090 web services has been done, showing a great performance in all the tests as the algorithm finds a valid solution with a short execution path.

P. Rodríguez-Mier · M. Mucientes (✉) · M. Lama · M. I. Couto
Department of Electronics and Computer Science, University of Santiago de Compostela, E-15782 Galicia, Spain
e-mail: manuel.mucientes@usc.es

P. Rodríguez-Mier
e-mail: pablo.rodriguez.mier@usc.es

M. Lama
e-mail: manuel.lama@usc.es

M. I. Couto
e-mail: miguelixen.couto@usc.es

Keywords Web services composition · Genetic programming · Evolutionary algorithms

1 Introduction

Web Services are interfaces that describe a collection of operations that are network-accessible through standardized web protocols, and whose features are described using a standard XML-based language [3, 11]. This includes functional features that indicate the input/output needed to invoke the execution of a web service; nonfunctional features such as cost, robustness, reliability, etc.; interaction features or choreography that describe how a client dialogs with the service in order to consume its functionality; and structural features or orchestration that model how the internal components of the service are combined to execute it.

In this way, as the characteristics are available through the interfaces, web services can be automatically discovered and invoked by extern programs (clients). When programs do not find a service with the required functionality (inputs and outputs), it is possible to compose a new service automatically. This composite service combines the functionalities of other services to get the desired outputs. This combination consists of a set of services that are executed in a sequence or in a set of workflow-like structures that control the execution of the services (specified through web services composition languages as OWL-S [17] or BPEL4WS [10]).

In the last years several papers have dealt with the composition of web services. Some approaches consider the composition problem as a planning problem of several actions (services) that operate on an initial state (inputs and preconditions) and generate an output state (postconditions)

[13, 14, 20–25, 27]. In these proposals, the planning techniques are blended with semantic reasoning to combine the outputs of some services with the inputs of others. The main drawback is that in these approaches the result of the composition is a sequence of services and, therefore, they do not take into account other control constructions that are part of the OWL-S or BPEL4WS models. In this way, this particular problem has a computational complexity much lower that those compositions that follow languages like OWL-S or BPEL4WS.

Other papers solve the composition of services with machine learning techniques like genetic programming [5, 9, 26, 28]. In these approaches, the minimum execution path needed to achieve a solution is not considered in the fitness function, and therefore optimal individuals are not assured. Furthermore, these proposals are validated with a low number of services and then the effectiveness of the proposed algorithms cannot be really evaluated.

In this paper we present a genetic programming algorithm that solves the problem of composition of web services. The algorithm uses a context-free grammar to limit the valid structures, takes into account the attributes updating, and minimizes both the number of services of the composite solution and the execution path needed to achieve the desired result. A full validation has been done in four different repositories: OWL-S TC [15], a hand-made repository with 1,000 services, and three program-generated repositories proposed for the 2008 Web Service Challenge of the EEE conference [6]. The behavior of the algorithm shows a great performance, as in all the cases a correct composition was found. This validation demonstrates the generality of the evolutionary algorithm, as it does not depend on the structure and features of a given repository.

The paper is structured as follows: Sect. 2 introduces the web services composition problem, and Sect. 3 describes the different approaches that have already been proposed. Then, Sect. 4 presents the proposed genetic programming-based algorithm for web services composition, Sect. 5 comments the obtained results and, finally, Sect. 6 points out the conclusions.

2 Problem description

In this paper, we consider that web services are only characterized by their functional features (that is, inputs and outputs), which are semantically described through ontologies. With this semantic description the output of a service O_{So} matches the input of other service I_{Si} when O_{So} is a subclass of I_{Si}. In general, when a concept C_i is a subclass of a concept C_j ($C_i \subseteq C_j$), then there is a semantic matching between C_i and C_j. This semantic matching will be used when two o more concepts are compared in the different stages of our algorithm.

Considering this description for web services, the composition problem can be formulated as the automatic construction of a workflow that coordinates the execution of a set of services that interact among them through their inputs and outputs (applying the semantic matching). This workflow, therefore, has services and a set of control structures that define both the behavior of the execution flow and the inputs/outputs of the services related to those structures. Thus typical control structures of web services composition languages are:

- *Sequence structure*, where the output of a service is the input of one of the following services of the sequence. This is the simplest control structure of the workflow languages.
- *Selection (choice) structure*, where an output can be achieved through two or more services, which therefore share the same output, but only one service will be selected and executed.
- *Parallel (split) structure*, where two o more services are executed in parallel and, as result, produce several and different outputs.
- *Parallel and synchronized (splitJoin) structure*, where the execution ending of services that run in parallel is synchronized. In this construction the services outputs are tipically different.
- *Loop structure*, where a set of services are executed until a given condition is verified. This structure does not impose any condition to the input/output concepts, although some approaches [5] assume that there must be a set of data in order to be individually used in each loop iteration.

These structures are shared by OWL-S[1] and BPEL4WS, which are the languages of the service repositories we have used to validate the proposed algorithm. In this sense it is important emphasize that the proposed algorithm is *independent* of the web services composition language, because the behavior of the control structures is defined in a general way.

As has been mentioned, the composition of web services consists of a set of services that are executed in a sequence or in a set of workflow-like structures that control the execution of the services. These two problems are very different from the point of view of the computational complexity:

[1] In OWL-S a *process* is used with the same meaning as a service. Thus a single service is named as an atomic process, and composite services are named as composite processes. We use this notation in the grammar that describes the chromosomes of our evolutionary algorithm.

– Sequence-based compositions: the complexity is $O(n!)$, where n is the size of the services repository[2].
– Workflow-like structures: the complexity is $O(n!\ t)$, where t is the number of different structures that can be generated. The structures can be represented by trees, and can be defined by a context-free grammar. Therefore, t depends on the grammar and on the maximum tree depth (d). If we assume that the grammar generates a complete binary tree[3], the maximum number of leaf nodes is 2^d. Thus, if the grammar has m different control structures, then the number of different structures that can be generated is $t \propto m^{2^d}$, as for each leaf node a different control structure can be selected. t is proportional to this expression because not all the rules in the grammar generate two internal nodes and/or not all the leaf nodes are control structures. This is the case for the context-free grammar defined in this paper (described in Sect. 4.1, Fig. 2). Finally, the complexity of workflow-like structures is $O\left(n!\ m^{2^d}\right)$.

As can be seen, workflow-like compositions have a much higher number of candidate solutions than sequence-based compositions, which makes classical search methods not applicable for this kind of web services composition. Figure 1 shows the size of the search space for both sequence and workow-like compositions and two different services repositories with sizes 100 and 8,000. The x-axis represents the depth of the tree for workflow-like compositions and the corresponding search space size has been calculated using the context-free grammar defined in this paper (Fig. 2). The size of the search space has been calculated in a precise way, generating all the valid structures for each tree depth and calculating the number of different compositions using the number of services of the corresponding services repository. Even if the number of services used for workow-like compositions is 80 times lower than that of a sequence-based approach (100 vs. 8,000), the size of the search space for a depth of five is larger for workflow-like compositions.

3 Related work

Web services composition has attracted widespread attention in recent years. Although there are several proposals to classify the approaches that focus on this topic [2, 12], in this paper we distinguish two kinds of algorithms depending on the complexity of the problem they solve:

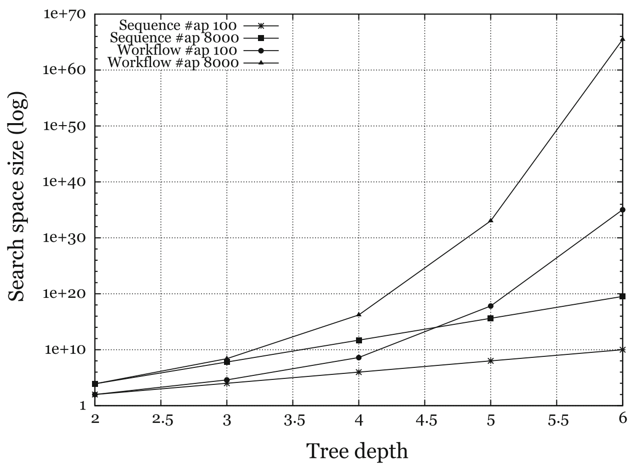

Fig. 1 Search space size of sequence and workflow-like structures for different services repository sizes

1. Algorithms that solve the problem of generating *services sequences* whose execution leads to the desired result.
2. Algorithms whose aim is to obtain a workflow composed by a set of control structures that coordinate the service execution. Usual control structures in most workflow languages are sequences, parallel executions, synchronizations, selections and loops. The complexity of this kind of services compositions is higher than the sequence generation (Fig. 1) because, to achieve a solution, they must be taken into account the dependencies among control structures and how each structure deals with input/output data.

3.1 Services sequence-based composition

An extensive research in services composition has been focused on planning-based approaches in which the composition is modeled as a planning problem [13, 19]. In these approaches there is an initial state defined by a set of both inputs and preconditions that the composite service must verify; a set of operators (or *services*) that are executed to obtain new and intermediate states; and a final state defined by a set of both outputs and postconditions that the solution must also verify. The composite service is therefore generated by a sequence of services whose ordered execution allows to achieve the requested outputs from the inputs.

Following this general model, different planners have been applied, such as graph analysis-based planners [14, 27], where the GraphPlan [8] algorithm is adapted to find

[2] This complexity is for the worst case: a composition which uses all the services of the repository. However, if we knew in advance the size of the composition (this is, in general, not truth), the complexity would be $O\left(\frac{n!}{(n-p)!}\right)$, where p is the number of services of the composition.

[3] In a complete binary tree every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

services compositions with optimal paths from inputs to outputs; logic-based planners [22], where the reasoning capabilities of a logic paradigm are used to obtain the services whose execution is compliant with the description of the state where they are applied to; hierarchical planners [16, 18, 25], where the hierarchical representation of composite services is considered to reduce the complexity in generating automatic sequences at different hierarchy levels; or planning as model checking [4, 7, 21], where the non-determinism and partial observability of services is managed for the generation of compositions. In these approaches, as for hierarchical planners, it is necessary to have an abstract representation of the workflow that models the composite service (with abstract service descriptions). The planner has to select the concrete services that better fit to the predefined compositions.

The main drawback of these approaches is their low performance when *the search space is huge*, that is, when the number of services and the input/output interactions among them is high. In this case the number of operators (services) that could be applied to a given state (verifying partially its inputs and preconditions) is high and, therefore, the number of potential intermediate states is huge. In this situation, finding a solution is a hard problem that requires the use of optimal search techniques. To deal with this issue some strategies have been proposed:

– In [20] the planning algorithm is combined with regression search to minimize the *number of services* that could be applied to a given state. Thus, once a services sequence is obtained, an heuristic greedy search is applied in a backward sense to approximate the optimal sequence of services.
– In [23] a query index with semantic information about inputs/outputs concepts of the services is created in order to reduce the reasoning time needed to obtain a matching between the inputs and outputs of the services.

The other disadvantages of these approaches are that: (1) they have not been validated in large services repositories; and (2) the generation of services sequences, usually, has not the optimal execution path, because parallel structures are not considered as part of the solution. However, as implicit loops are allowed in the algorithm, a solution to this issue would be to apply a pattern matching algorithm to discover control structures in the sequences [14].

3.2 Automatic workflow composition

Several approaches based on evolutionary algorithms have been proposed to obtain services compositions whose description is carried out through workflows [5, 9, 26, 28]. For example, [5] describes an algorithm for services composition in BPEL that follows a similar approach to the one presented in this paper. The main differences with our proposal are that: (1) it does not show a formal description of the grammar to compound services; (2) attributes updating after crossover and mutation is not explicitly managed. Therefore, it is difficult to evaluate to which degree all the interactions among services are fulfilled to get a correct solution; (3) minimum execution paths are not assured because this parameter is not included as part of the fitness function; and (4) the algorithm has been validated in a private repository.

In [9], authors consider a workflow of tasks and a set of services that may execute each of those tasks. The proposal presents an evolutionary algorithm to associate a task with an optimal service, and considers a fitness function implemented as a multi-objective and distance-based algorithm that evaluates quality of service parameters. In this algorithm, therefore, it makes no sense to include the minimum execution path of the composite service as a criterion to select individuals, because the workflow is predefined. A similar approach has been presented in [26], where the fitness function is calculated as a formula with weights for the different quality of service parameters.

In [28] a particle swarm optimization algorithm is applied to optimize the selection of services that are part of the solution. In order to do that, the semantic similarity between the service characteristics is calculated, obtaining a set of measures (or distances) that define the relation between a service and the other services of the repository. When these measures are available, the algorithm obtains a services sequence with optimal distances among the services. This work has not been validated in a large repository; it used the Amazon services to demonstrate the viability of the algorithm.

Furthermore, in the bibliography many other approaches for composition of service workflows have been proposed, approximating the solution with different search strategies such as heuristic search [1] or graph analysis [29]. Common drawbacks of these proposals are that they cannot manage all the control structures as are defined in workflow languages as BPEL4WS and OWL-S, and the performance of the algorithm decreases as the number of services and interactions among them is huge.

With this state of the art, we can conclude that the main differences between other approaches and our proposal are:

– Current approaches focusing on automatic generation of workflows do not consider all the control structures of the workflow languages like OWL-S and BPEL4WS. Thus, planning algorithms only obtain services sequences and evolutionary or optimization techniques do not manage the complete set of workflow-like structures.

- Some approaches do not minimize the execution path needed to execute the composite service, that is, they do not maximize the use of parallel control structures to reduce execution times.
- Existing proposals have not been validated in several repositories with different features in order to demonstrate the generality of the algorithm.

All these drawbacks have been tackled by our genetic programming-based approach to web services composition, which is described in the next section.

4 Genetic programming for web services composition

Web services composition requires the combination of many atomic services using several control structures. This combination of elements can be modeled, in a natural way, with a tree that represents the solution to a web services composition. As not all the combinations of atomic services and control structures are valid from a syntactical point of view, restrictions in the syntactical structure of a solution (web services composition) can be described with a context-free grammar. Genetic programming is especially adequate for web services composition due to:

- Genetic programming can deal with solutions with very different structures as the individuals are usually represented by trees and, moreover, the trees can have different depths and number of nodes.
- A context-free grammar can be naturally included to generate new individuals, and to produce right structures for the individuals after crossover and mutation.
- Web services composition has a hierarchical structure, i.e., several atomic services generate a composite service, several composite services produce a more complex composition and so on, until the desired solution is found. Therefore, the subtrees of a tree represent simple compositions that contribute to the solution. Intermediate compositions can be interchanged between trees, in order to improve the performance of the new trees (solutions). This is exactly what is implemented with the crossover operator in genetic programming.

The first step in the design of an algorithm for web services composition requires the definition of the type of composite services that are going to be build. A compact definition of the valid structures of a tree (chromosome) for a web services composition can be described by a context-free grammar.

4.1 Context-free grammar

A context-free grammar is a quadruple (V, Σ, P, S), where V is a finite set of variables, Σ is a finite set of terminal symbols, P is a finite set of rules or productions, and S is an element of V called the start variable. The grammar that defines the valid structures for web services composition is described in Fig. 2. The first item enumerates the variables, then the terminal symbols, in third place the start variable is defined, and finally the rules for each variable are enumerated. When a variable has more than one rule, rules are separated by symbol "|".

The grammar has been defined to fulfill the syntax of the most common web services composition languages (OWL-S and BPEL4WS), and is completely independent of the services repository. *<initialProcess>* is the start variable of the grammar and generates an atomic or a composite process.

Variable *<process>* defines either composite processes or atomic processes. Two of the four rules of this variable are recursive and, therefore, a process can be composed of any number of atomic and composite processes. Finally, variable *<compositeProcess>* represents the combination of a control structure and two processes (of any type), i.e., a composition of at least two processes.

All the nodes of type variable, together with terminal symbol *atomicProcess* constitute the service nodes. They are characterized by the following attributes:

- Control structure: the node of type control structure ({*choice*, *sequence*, *split*, *splitJoin*}) of which the service node depends on. The control structure manages the interaction among the services that share that control.
- Available inputs: are those inputs available for a service. A subset of them are selected as inputs to the service. An input can be available in two ways. First, if the user introduces that input. In second place, if a service that belongs to the composition and has been executed before (in the composition flow), generates as output that service functionality.

- $V = \{initialProcess,\ process,\ compositeProcess\}$
- $\Sigma = \{atomicProcess,\ choice,\ sequence,\ split,\ splitJoin\}$
- $S = initialProcess$
- Rules:
  - $<initialProcess>\ ::=\ <compositeProcess>\ |\ atomicProcess$
  - $<process>\ ::=\ <compositeProcess>\ <process>\ |\ atomicProcess\ <process>\ |\ <compositeProcess>\ |\ atomicProcess$
  - $<compositeProcess>\ ::=\ choice\ <process>\ <process>\ |\ sequence\ <process>\ <process>\ |\ split\ <process>\ <process>\ |\ splitJoin\ <process>\ <process>$

Fig. 2 Context-free grammar for web services composition

- Necessary inputs: are the inputs that the node needs for running all the atomic processes in the subtree for which the node is the root node. These inputs or their subclasses have to be provided by the user or by other services of the composition.
- Obligatory inputs: in some situations, the outputs of several services have to be used as inputs to the current service. This means that at least one of those outputs has to be selected as input to the current service (a semantic matching among them must exist). An example of this situation is the sequence of two services S_a and S_b. Let $O_a = \{o_1^a, \ldots, o_{n_a}^a\}$ be the set of outputs generated by service S_a, and $I_b^n = \{i_1^b, \ldots, i_{n_b}^b\}$ be the set of necessary inputs of S_b. Then, $O_a \cap I_b^n \neq \emptyset$. If this condition is not fulfilled, the composition of services S_a and S_b is not a sequence, and the structure is not valid. Therefore, the inputs of the service must contain a subset of the obligatory inputs. Following the example, the obligatory inputs of service S_b are the outputs of service S_a, i.e., $I_b^o = O_a$.
- Outputs: generated by the service. They can be directly generated by the service (if it is an atomic process) or by the subtree with the service as root node (composite process).

4.2 Attributes updating

The initialization of a tree (web services composition), or a modification of it due to crossover or mutation, requires the updating of all the attributes of each node. The initial step of the algorithm resets all the attributes of all the nodes in the tree, and then initializes the necessary inputs of the root node (<*initialProcess*>) to the set of inputs of the web services composition to be solved. Then, the tree is traversed in preorder, updating the attributes of each node. To traverse a tree in preorder, the following operations must be performed recursively at each node, starting with the root node: first, visit the root. Then, traverse the subtrees that have as root node the children of the root. Children are traversed in order, starting with the leftmost node and continuing to the right. Updating the attributes of each node is done in a different way depending on the type of attribute:

- Control structure (cs): this attribute is propagated in a top-down way. This means that a node inherits the attribute value from its parent. There is an exception to this rule. The node will set its control structure to its leftmost brother when that brother is a control structure.
- Available inputs (I^a): they are propagated in a top-down way. If an input is available for a node, it will also be available for all its children. When the control structure of the node is *sequence*, all the outputs of the brothers to the left of the node will also be added as available inputs.
- Necessary inputs (I^n): the propagation is done in a bottom-up way. This means that, if and only if the node is a leaf node, all its ancestor nodes will add as necessary inputs the necessary inputs of the node.
- Obligatory inputs (I^o): they are propagated in a top-down way. When the control structure of the node is *sequence* and the brother node immediately to the left is a service node, the obligatory inputs will be set with the following algorithm:
-
  1. Traverse in preorder the subtree that has as root node the brother node just to the left of the current node (the one for which the obligatory inputs are being calculated).
  2. Get the last node traversed in that process. It will be the rightmost node of the subtree.
  3. If both the last and current nodes depend on the same control structure (they have a reference to the same node of type *controlStructure*), then the outputs of the last node will be the obligatory inputs of the current node.
  4. Else, the outputs of the brother node immediately to the left will be the set of obligatory inputs of the current node.
- Outputs (O): the attribute is propagated in a bottom-up way (the outputs of a node will also be outputs of its parent), except when the leftmost child of the node is a *choice* control structure. Outputs for this situation are obtained as: $O = O_1 \cap \ldots \cap O_n$, i.e., the intersection of the outputs of all the children of the node.

4.2.1 An example

Figure 3 shows a services composition. Terminal symbols (leaves of the tree) are represented by rectangles or squares, and variables are shown as flatted circles. Each node includes the values of the different attributes: the control structure governing the node (cs), the available inputs (I^a), the necessary inputs (I^n), the obligatory inputs (I^o) and the outputs (O). In this example the initial available inputs are i_a and i_b, and the outputs required to solve the composition are $o_{6.1}$ and $o_{6.2}$.

Attributes updating starts from the root node, traversing the tree in preorder. When the first atomic process node (3.1) is reached, its available and obligatory inputs are set to its parent values, which were also taken from its ancestor (top-down updating). This service uses i_a as input and generates $o_{3.1}$ as output. Therefore, the necessary inputs

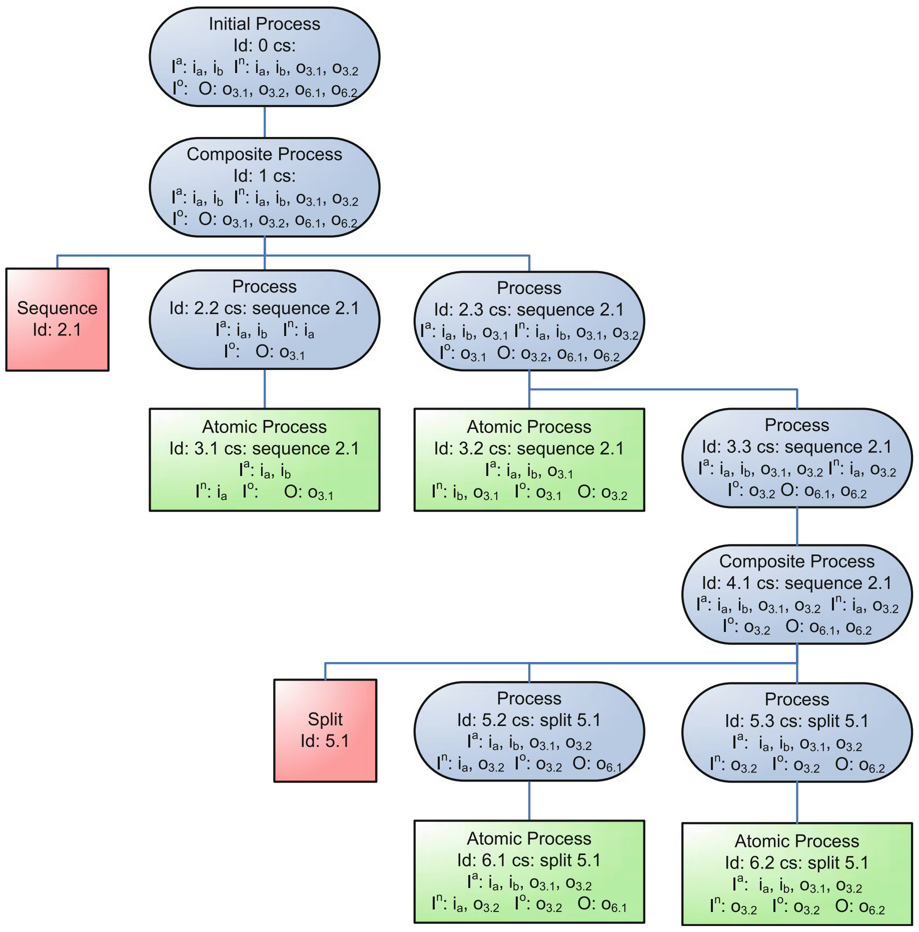
Initial Process
Id: 0 cs:
I^a: i_a, i_b I^n: i_a, i_b, $o_{3.1}$, $o_{3.2}$
I^o: O: $o_{3.1}$, $o_{3.2}$, $o_{6.1}$, $o_{6.2}$
Composite Process
Id: 1 cs:
I^a: i_a, i_b I^n: i_a, i_b, $o_{3.1}$, $o_{3.2}$
I^o: O: $o_{3.1}$, $o_{3.2}$, $o_{6.1}$, $o_{6.2}$
Sequence
Id: 2.1
Process
Id: 2.2 cs: sequence 2.1
I^a: i_a, i_b I^n: i_a
I^o: O: $o_{3.1}$
Process
Id: 2.3 cs: sequence 2.1
I^a: i_a, i_b, $o_{3.1}$ I^n: i_a, i_b, $o_{3.1}$, $o_{3.2}$
I^o: $o_{3.1}$ O: $o_{3.2}$, $o_{6.1}$, $o_{6.2}$
Atomic Process
Id: 3.1 cs: sequence 2.1
I^a: i_a, i_b
I^n: i_a I^o: O: $o_{3.1}$
Atomic Process
Id: 3.2 cs: sequence 2.1
I^a: i_a, i_b, $o_{3.1}$
I^n: i_b, $o_{3.1}$ I^o: $o_{3.1}$ O: $o_{3.2}$
Process
Id: 3.3 cs: sequence 2.1
I^a: i_a, i_b, $o_{3.1}$, $o_{3.2}$ I^n: i_a, $o_{3.2}$
I^o: $o_{3.2}$ O: $o_{6.1}$, $o_{6.2}$
Composite Process
Id: 4.1 cs: sequence 2.1
I^a: i_a, i_b, $o_{3.1}$, $o_{3.2}$ I^n: i_a, $o_{3.2}$
I^o: $o_{3.2}$ O: $o_{6.1}$, $o_{6.2}$
Split
Id: 5.1
Process
Id: 5.2 cs: split 5.1
I^a: i_a, i_b, $o_{3.1}$, $o_{3.2}$
I^n: i_a, $o_{3.2}$ I^o: $o_{3.2}$ O: $o_{6.1}$
Process
Id: 5.3 cs: split 5.1
I^a: i_a, i_b, $o_{3.1}$, $o_{3.2}$
I^n: $o_{3.2}$ I^o: $o_{3.2}$ O: $o_{6.2}$
Atomic Process
Id: 6.1 cs: split 5.1
I^a: i_a, i_b, $o_{3.1}$, $o_{3.2}$
I^n: i_a, $o_{3.2}$ I^o: $o_{3.2}$ O: $o_{6.1}$
Atomic Process
Id: 6.2 cs: split 5.1
I^a: i_a, i_b, $o_{3.1}$, $o_{3.2}$
I^n: $o_{3.2}$ I^o: $o_{3.2}$ O: $o_{6.2}$

Fig. 3 A chromosome representing the composition of several atomic processes

and the outputs will be set to these values and propagated to all the ancestors of the node.

Following the preorder traversal, node 2.3 is visited. As this node has a *sequence* control structure and has brother service nodes on the left, both the available and obligatory inputs require a different updating. The available inputs are those inherited from the parent (top-down updating) plus all the outputs generated by the brother nodes to the left, i.e., $o_{3.1}$ is added as an available input. On the other hand, the obligatory inputs are the outputs of the brother node ($o_{3.1}$). These attribute values are propagated down to node 3.2. This node is an atomic process that generates output $o_{3.2}$ using inputs i_b and $o_{3.1}$. Attributes outputs and necessary inputs are consequently updated and propagated to its ancestors.

The next traversed node is 3.3. Again, this node has a *sequence* control structure and has brother service nodes on the left. Therefore, the output of node 3.2 is added as available input to the node and, also, the obligatory inputs attribute is set to this value.

Both the available and obligatory inputs are propagated down. Thus, nodes 6.1 and 6.2 have to use $o_{3.2}$ as input. Both nodes propagate up the necessary inputs (i_a, $o_{3.2}$) and the outputs ($o_{6.1}$, $o_{6.2}$), and the updating process ends with the configuration shown in Fig. 3.

4.3 Genetic programming-based algorithm

Figure 4 describes the genetic programming algorithm that has been used for web services composition. The first three steps of the algorithm correspond to an initialization. t represents the number of iterations, while *timesRun* will be used to detect situations in which the search gets stuck. The iterative part of the algorithm starts at step four. This part will be repeated until the maximum number of iterations is reached or the best possible solution is found. The main stages of the iterative part are the selection of the individuals, the crossover and mutation to generate new individuals, the post-processing, their evaluation, the replacement of the population, the local search, and the checking of stuck situations in the search process. All of them are described in detail in the next sections.

4.3.1 Initialization

The first step of the algorithm is the generation of the initial population. A new individual is generated applying randomly the rules of the grammar. If the depth of the tree reaches the maximum predefined value, then all the nodes of type service at that depth are transformed to

```
1. Initialize population
2. Evaluate population
3. $t = 1$, $timesRun = initialTimesRun$
4. While $t \leq maxT$ and $fitness_{best} < maxFitness$
   (a) Selection
   (b) Crossover
   (c) Mutation
   (d) Post-processing
   (e) Evaluate new individuals
   (f) Replace population
   (g) Run the local search
   (h) $t = t + 1$
   (i) If $bestInd(t) = bestInd(t-1)$, then $timesRun = timesRun - 1$
   (j) If none of the individuals of the population have been
       created in the current iteration, then $timesRun = timesRun - 1$
   (k) If $timesRun < 0$, then:
       i. Reinitialize population, keeping only the best individual.
       ii. Evaluate new individuals
       iii. $timesRun = initialTimesRun$
5. Final post-processing
```

Fig. 4 Genetic programming algorithm for web services composition

atomicProcess nodes. Once the structure of the tree has been defined, the attributes of the nodes must be initialized using the algorithm defined in Sect. 4.2.

This attributes updating algorithm is run with one special characteristic. When an *atomicProcess* node is reached during the traversal of the tree, as no specific service has been assigned to it, one has to be selected from the repository. The selection is done randomly from the set of services that fulfill: $I_j^a \supseteq I_k$ and $I_j^o \cap I_k \neq \emptyset$. Thus, a service k can be selected if its inputs are a subset of the available inputs of the *atomicProcess* node j (I_j^a) and if at least one of the inputs of k belongs to the set of obligatory inputs of j (I_j^o).

4.3.2 Evaluation

The calculation of the fitness of each individual of the population is done analyzing four criteria: generated outputs, used inputs, execution time of the composite service and number of nodes of type *atomicProcess*:

$$fitness = \omega_1 \cdot \left(\frac{\sum_i^{|O_{obj}|} \frac{1}{DO_i + 1}}{|O_{obj}|} + \frac{|I_{root}^n \cap I_{obj}|}{|I_{obj}|} \right) + \omega_2 \cdot \frac{1}{runPath} + \omega_3 \cdot \frac{1}{\#atomicProcess} \quad (1)$$

where O_{obj} are the outputs that are required to solve the composition, DO_i is the distance of the individual to the ith required output, I_{root}^n are the necessary inputs of the root node (this node is the result of the composition of the services), I_{obj} are the inputs provided to solve the composition, *runPath* is the execution time of the composite service, *#atomicProcess* is the number of atomic processes in the tree, and ω_k are values that weight the importance of each criterion.

The first and second criteria indicate the degree to which a valid solution has been found. The first one is the number of outputs (or subclasses of them), of those that were required, that have been generated by the composition. The second criterion is the number of inputs (or superclasses of them), of those provided by the user, that have been used.

In order to guide the composition, the use of a crisp criterion for the outputs is not adequate, and the concept of distances to outputs (DO_i) must be introduced. For example, if the expected result of a composition is a sequence of ten services, and the desired output is provided by the last service, a composition of the first nine services will not generate the desired output and, therefore, with a crisp criterion, this part of the fitness function would be evaluated as 0. However, if the fitness function measures the distance between the composite service (of nine atomic processes) and the desired output, it will reflect that the composite process is close to find a valid solution (only a new atomic service needs to be added). The distance of an individual to the ith desired output (DO_i) is calculated in the following way:

$$DO_i = \min_j DO_{ij} \quad (2)$$

where DO_{ij} is the distance of the jth atomic service of the individual to the ith output. Thus, the distance of the individual to the output is the minimum of the distances of its atomic services to that output. Also,

$$DO_{ij} = \min_k DO(S_j, S_k) : o_i \in O_k \quad (3)$$

where S_j and S_k are services, O_k is the set of outputs generated by service S_k, and o_i is the ith output. DO is the distance between two atomic services, and is defined as the minimum number of atomic services that need to be composed in sequence, starting with S_j, in order to generate an output of S_k. For example, if $O_j \cap O_k \neq \emptyset$, then $DO(S_j, S_k) = 0$.

The third criterion is the execution time of the composite service. This time depends on the execution time of each atomic service but, also, on the control structures in the following way:

– *sequence*: the execution time is the sum of the times of all the services in the sequence.
– *split* and *splitJoin*: the execution time is equal to the time of the slowest service belonging to this control structure, as all the services are executed in parallel.

- *choice*: in this control structure, only one service of the composition is executed. As the selected service is only known at run time, the worst time of all the services in the *choice* composition has to be selected. Therefore, the execution time is calculated in the same way as for the *split* control structure.

Finally, the last criterion is related with the complexity of the composite service. The higher the number of atomic processes in the composition, the higher the complexity.

4.3.3 Selection

The selection mechanism that has been used is the binary tournament selection. In a k-tournament selection, k individuals are randomly picked from the population with replacement, and the best of them is selected. In this case, $k = 2$ (binary tournament selection).

4.3.4 Crossover

The crossover operator replaces a subtree of an individual with a subtree of other individual. The process is as follows:

- Select randomly a node of type service in the first individual.
- Generate the set of candidate nodes in the second individual. These nodes must have the following characteristics:
  - They must be of type service.
  - $\left(I_2^n - O_2\right) \cap I_1^o \neq \emptyset$. $I_2^n - O_2$ represents all the inputs that are used by the subtree of the second individual and that have not been generated inside that subtree. This set of inputs must contain at least one of the obligatory inputs (or their subclasses) of the subtree that is going to be replaced in the first individual.
  - $I_2^n - O_2 \subseteq I_1^a$. Also, the set of inputs used in the subtree of the second individual must be a subset of the available inputs (or their subclasses) for the subtree of the first individual.
  - Select randomly a node of the candidate nodes set, and replace the subtree of the first individual with the selected subtree of the second individual.
  - Execute the attributes updating algorithm. During the execution of the algorithm, if a leaf node of type *atomicProcess* is reached, two conditions must be checked: $I_j^a \supseteq I_k$ and $I_j^o \cap I_k \neq \emptyset$, i.e., the inputs of the process (I_k) must be a subset of the available inputs (or subclasses of them) of the node and, also, they must contain at least one of the obligatory inputs (or their subclasses) of the node. If the conditions are not fulfilled, a new atomic process must be selected using the same procedure as in the initialization stage (Sect. 4.3.1).

4.3.5 Mutation

The mutation operator modifies a subtree of the individual. First, a node must be randomly selected. If the node is of type variable, then the subtree that has as root the selected node is eliminated. The new subtree is generated applying the rules of the grammar randomly for that variable in the same way as in the initialization stage (Sect. 4.3.1). On the other hand, if the node is of type terminal, there are two cases:

- If the node is a control structure, a new one is randomly selected.
- If the node is an atomic process, there are two posibilities:
  - A new process is randomly selected from the repository using the same conditions defined in the initialization stage (Sect. 4.3.1).
  - The node is substituted with a process node and a subtree is generated applying the rules of the grammar in the same way as for nodes of type variable.

In all the cases, the attributes updating algorithm must be run. Also, the validity of the atomic processes must be checked and, if necessary, a new selection of the atomic processes is done.

4.3.6 Replacement

The selection mechanism is a population-based selection approach, i.e., parents and their corresponding offspring are combined generating a population with a size $2N$ (being N the size of the initial population), and the best N individuals are selected for the next population.

4.3.7 Post-processing

The size and complexity of the trees representing the individuals has to be managed in order to improve the search, reduce the time per iteration and, also, to simplify the final composite service. The post-processing stage consists of four steps that are executed at the end of the algorithm. Moreover, two of these steps are also executed at the end of each iteration. The steps must be executed in the following order:

1. *Eliminate useless atomic services*: an atomic service is useless if none of their outputs neither contribute to the

objective outputs nor are inputs to other services. Elimination of this kind of services is recursive, i.e., it is repeated while in the previous step a service was eliminated. New useless services can appear due to the elimination of a useless service. This procedure is executed at the end of the algorithm.
2. *Eliminate useless control structures*: a control structure is useless when only one atomic service depends on that control structure. A control structure is used to compose services and, therefore, a minimum of two services are needed. Useless control structures have to be eliminated, and the atomic process belonging to it is assigned to the control structure of its closer ancestor. This step needs to be done at the end of each iteration.
3. *Eliminate consecutive and equal control structures*: when a node and its parent have the same type of control structure and it is an *split* or an *splitJoin*, both control structures can be merged. This step is executed only at the end of the algorithm.
4. *Tree flatten*: the depth of the trees is limited in order to prevent an infinite growth of the individuals. The proposed context-free grammar is unambiguous, and this means that an individual can be only represented by a tree. However, due to crossover, mutation and, also, due to the recursive rules in the grammar, the number of nodes and the depth of the trees can grow at a high rate. In large trees, usually some of the internal nodes are useless, i.e., they could be deleted without modifying the fenotype of the individual (the composition remains equal), although the genotype is changed. This process generates smaller individuals, which improves the search for better compositions. Tree flatten transforms a tree in its equivalent with the minimum depth, and it is done in each iteration. To keep the same fenotype, it is necessary to respect the precedence among the different control structures in the individual. The process is started in the leaf nodes and ends when the root node is reached. Each node in the tree is gone up to the depth of its control structure node.

4.3.8 Local search

The objective of the local search is to improve some of the individuals of the population implementing a search process with a low degree of exploration and a high degree of exploitation, i.e., a very exhaustive search in the neighborhood of the individual. In this case, the local search has been applied to only one individual of the current population: the best individual. If the local search was already run for that individual in previous iterations, then a new individual is randomly selected to execute the local search.

The local search algorithm is described in Fig. 5. It is a greedy algorithm (proceeds by changing the current assignment by always trying to increase the fitness) called steepest ascent hill climbing [24]. This algorithm fulfills two requirements that are necessary for its adequate cooperation with the genetic programming algorithm for web services composition: it makes a complete search in the neighborhood of the solution until it finds the local maximum, and it is very fast. A loop (lines 2–14) is run until the local search fails to improve the best solution in one iteration. The best solution of the iteration (Ω') is initially set to the best solution (Ω). For all the neighbors of the the best solution, the best one is picked if it improves the best solution of the iteration (lines 5–9). Finally (lines 10–13), if the best solution of the iteration improves the best solution, the best solution is updated and the local search continues. Else, the best solution is returned and it will replace the original solution in the population if it is better.

The local search requires the generation of the neighbors of an individual (line 5). The neighborhood can be obtained substituting each atomic process of the individual with another atomic process from the repository (fulfilling some conditions that will be detailed later). As there can be several candidates for each replacement and there are also several *atomicProcess* nodes in the individual, making all the combinations can generate a huge number of neighbors. Thus, in order to speed up local search, a reduced number of neighbors will be generated as follows:

1. Select randomly the number of *atomicProcess* nodes of the individual that will be modified ($\#ap_{LS}$).
2. Pick randomly those *atomicProcess* nodes that will be modified.
3. For each node of type *atomicProcess* (ap_j) that has been picked, look for all the processes in the repository that fulfill the following conditions:

```
1: Obtain the initial solution, $\Omega$.
2: repeat
3:     $\Omega' = \Omega$
4:     $continue = false$
5:     for all neighbors $\Omega''$ of $\Omega$ do
6:         if $fitness(\Omega'') > fitness(\Omega')$ then
7:             $\Omega' = \Omega''$
8:         end if
9:     end for
10:    if $fitness(\Omega') > fitness(\Omega)$ then
11:        $\Omega = \Omega'$
12:        $continue = true$
13:    end if
14: until $continue$
15: Return $\Omega$
```

Fig. 5 Steepest ascent hill climbing algorithm [24]

(a) $I_j^a \supseteq I_{jk}^n$
(b) $I_j^o \cap I_{jk}^n \neq \emptyset$
(c) $O_{jk} \supseteq O_j^n$
where I_{jk}^n are the necessary inputs of process ap_{jk}, $k = 1,\ldots, \alpha_j$. ap_{jk} is the kth atomic process of the repository that fulfills the conditions for node j (which corresponds to atomic process ap_j), O_{jk} are the outputs of ap_{jk}, and O_j^n are the outputs that were generated by atomic process ap_j and that were used as inputs by other atomic processes of the individual.

4. Calculate for each ap_{jk} the probability to be selected: $p_{jk} = \eta\ fitness_{jk}$, where $fitness_{jk}$ is the fitness of the individual after the replacement of the atomic process of node j by ap_{jk}, and η is a normalization factor.
5. For each considered ap_j, pick randomly one of the ap_{jk} using the calculated probabilities (p_{jk}).
6. The neighborhood is composed of all the individuals obtained after replacing or keeping the corresponding nodes (ap_j). The size of this neighborhood is $2^{\#ap_{LS}} - 1$.

4.3.9 Reinitialization

The last steps of each iteration (Fig. 4) update the value of *timesRun*, decreasing it when the best individual has not improved and also when no individuals of the current iteration have survived the replacement process. If *timesRun* takes a value below 0, the population is reinitialized in the same way as in the initialization stage, but keeping the best individual.

5 Results

The validation of the genetic programming algorithm for web services composition has been done with a set of experiments with different degrees of complexity. Four different repositories have been used for test:

1. *OWL-S TC V2.2*, with 1,000 services described with the OWL-S profile[4].
2. Web Service Challenge 2008 (*WSC 2008*) repository 1, with 158 services represented in WSDL and whose inputs and outputs are semantically described[5].
3. Web Service Challenge 2008 (*WSC 2008*) repository 2, with 558 services represented in WSDL and whose inputs and outputs are semantically described.
4. Web Service Challenge 2008 (*WSC 2008*) repository 5, with 1,090 services represented in WSDL and whose inputs and outputs are semantically described.

The services compositions that have been tested are shown in Figs. 6, 7 and 8. For each example, a short description of the task that the composite service solves is given. Also, the available inputs (those provided by the user) and the desired outputs are enumerated. Then, the solution to the requested service is indicated: it is a combination of control structures and atomic processes. In most of the cases there are a few possible best solutions, but only one has been indicated in Figs. 6, 7 and 8. Finally, each of the atomic processes that are part of the solution are described. It should be noticed that, as the inputs and outputs of the repositories *WSC 2008* are semantically described, the names of the inputs and outputs of the atomic processes (Figs. 7 and 8) do not match up. For example, the output of a service in a *sequence* could not be an input to the next service. This is because the input of the next service is a superclass of the previous output (semantics has to be taken into account).

Table 1 shows the results for all the test examples described in Figs. 6, 7 and 8. Each row in the table represents the results of the evolutionary algorithm for a test example. As evolutionary algorithms are nondeterministic, the result of one run over an example is not meaningful. Thus, for each of them 10 runs were executed. The columns represent the time to obtain the best solution found by the algorithm, the percentage of provided inputs that have been used by the atomic processes, the degree of fulfillment of the required outputs, the fitness value, the execution time (*runPath*) of the composite service (the execution time of each atomic service has been established to 1) and the number of atomic processes of the tree. For each of these columns, two values are represented: $\overline{\chi}$ is the arithmetic mean over 10 runs, and σ is the standard deviation over the 10 runs, which reflects the robustness of the probabilistic algorithm to obtain similar results regardless the followed pseudo-random sequence.

The values that have been used for the parameters of the evolutionary algorithm are: $maxT = 100$, $initialTimesRun = 20$, population size = 200, crossover probability = 0.9, mutation probability = 0.03 (per gene), maximum depth of the tree = 9, $\omega_1 = 0.45$, $\omega_2 = 0.05$, $\omega_3 = 0.05$, percentage of the individuals to apply local search = 0.5%.

The first thing that must be noticed is that the fitness is, in nearly all the cases, under 1, as the execution time of the composite service and the number of atomic processes in the tree are greater than one (Eq. 1). The performance of the algorithm is good, as in all the tests an acceptable solution has been found for all the runs. This means that $I_{root}^n \cap I_{obj} = I_{obj}$ and $O_{root} \cap O_{obj} = O_{obj}$. Also, the search

[4] http://projects.semwebcentral.org/frs/download.php/386/owls-tc2_2_rev_2.zip.
[5] http://cec2008.cs.georgetown.edu/wsc08/downloads/ChallengeResults.rar.

1. Obtain the time interval and the diagnostic process for a hospital:
   - Inputs: _HOSPITAL
   - Outputs: _TIMEINTERVAL, _DIAGNOSTICPROCESS
   - Solution: HOSPITAL_DIAGNOSTICPROCESSTIMEINTERVAL_SERVICE
   - List of atomic processes:
     - HOSPITAL_DIAGNOSTICPROCESSTIMEINTERVAL_SERVICE:
       - Inputs: _HOSPITAL
       - Outputs: _TIMEINTERVAL, _DIAGNOTICPROCESS
2. Confirm if, given a town, country and a price, it is possible to buy coffee and whiskey:
   - Inputs: _COUNTRY, _TOWN, _RECOMMENDEDPRICE
   - Outputs: _COFFEE, _WHISKEY
   - Solution: sequence(TOWNCOUNTRY_HOTEL_SERVICE, HOTELRECOMMENDEDPRICE_COFFEEWHISKEY_SERVICE)
   - List of atomic processes:
     - TOWNCOUNTRY_HOTEL_SERVICE:
       - Inputs: _COUNTRY, _TOWN
       - Outputs: _HOTEL
     - HOTELRECOMMENDEDPRICE_COFFEEWHISKEY_SERVICE:
       - Inputs: _RECOMMENDEDPRICE, _HOTEL
       - Outputs: _COFFEE , _WHISKEY
3. Obtain the maximum price of a book given the academic item number of the author:
   - Inputs: _ACADEMIC-ITEM-NUMBER
   - Outputs: _MAXPRICE, _BOOK
   - Solution: sequence(ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE, AUTHOR_BOOKMAXPRICE_SERVICE)
   - List of atomic processes:
     - ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE:
       - Inputs: _ACADEMIC-ITEM-NUMBER
       - Outputs: _AUTHOR , _BOOK
     - AUTHOR_BOOKMAXPRICE_SERVICE:
       - Inputs: _AUTHOR
       - Outputs: _MAXPRICE , _BOOK
4. Get the maximum price of a book, its type and the recommended price in dollars using the academic item number of the author:
   - Inputs: _ACADEMIC-ITEM-NUMBER
   - Outputs: _MAXPRICE , _BOOK-TYPE , _RECOMMENDEDPRICEINDOLLAR
   - Solution: sequence(ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE, split(AUTHOR_BOOKMAXPRICE_SERVICE, BOOK_RECOMMENDEDPRICEINDOLLAR_SERVICE, BOOK_AUTHORBOOK-TYPE_SERVICE))
   - List of atomic processes:
     - ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE:
       - Inputs: _ACADEMIC-ITEM-NUMBER
       - Outputs: _AUTHOR , _BOOK
     - AUTHOR_BOOKMAXPRICE_SERVICE:
       - Inputs: _AUTHOR
       - Outputs: _MAXPRICE , _BOOK
     - BOOK_RECOMMENDEDPRICEINDOLLAR_SERVICE:
       - Inputs: _BOOK
       - Outputs: _RECOMMENDEDPRICEINDOLLAR
     - BOOK AUTHORBOOK-TYPE_SERVICE:
       - Inputs: _BOOK
       - Outputs: _BOOK-TYPE
5. Get the weather, map and hotel given the city:
   - Inputs: _CITY, _DURATION, _COUNTRY
   - Outputs: _WHEATHERSEASON, _MAP, _HOTEL
   - Solution: split(CITYCITY_MAP_SERVICE, CITY_WHEATHERSEASON_SERVICE, DURATIONCOUNTRYCITY_HOTEL_SERVICE)
   - List of atomic processes:
     - CITYCITY_MAP_SERVICE:
       - Inputs: _CITY
       - Outputs: _MAP
     - CITY_WHEATHERSEASON_SERVICE:
       - Inputs: _CITY
       - Outputs: _WHEATHERSEASON
     - DURATIONCOUNTRYCITY_HOTEL_SERVICE:
       - Inputs: _CITY, _DURATION, _COUNTRY
       - Outputs: _HOTEL

Fig. 6 Description of the web services compositions used for testing on repository *OWL-S TC V2.2*

times[6] that have been obtained are quite low, which is specially important for web services composition, as users require a fast answer to their query. Going into the details for each test:

- *OWL-S TC V2.2*–1: this test is very simple, as it is just an atomic process and not a services composition. However, it has been included to verify that also under simple conditions the algorithm works properly (the best fitness was always reached). The number of atomic processes is always the right one, while the depth is always the minimum possible one.
- *OWL-S TC V2.2*–2: in this example all the executions reached the best possible services composition (two atomic processes connected in a *sequence*).
- *OWL-S TC V2.2*–3: this example is similar to the previous one (a *sequence* of two services). The best values for all the objectives (inputs, outputs, execution time, and number of atomic processes) have been reached in all the runs.
- *OWL-S TC V2.2*–4: this composition requires the use of two nested control structures: a *sequence* and an *split*. This solution cannot be constructed with sequence-based compositions. The execution time of the composite process was most of the times the lower one (it was over only two times). The number of atomic processes has a higher variability, indicating that

[6] These times have been obtained with an Intel Xeon(R) Quadcore E5320 1.86GHz processor with 8GB of RAM, and the algorithm was implemented in Java and run on Linux.

1. Web Service Challenge testset 1:
   - Inputs: con1233457844, con1849951292, con864995873
   - Outputs: con1220759822, con2119691623
   - Solution: sequence(splitJoin(serv75024910, serv1599256986, serv1668689219), serv976005395, serv283321609, splitJoin(serv1738121452, serv1114869861), serv1876985918, split(serv1184302094, serv491618308))
   - List of atomic processes:
     - serv75024910
       - Inputs: con1653328292, con1849951292, con241744282
       - Outputs: con1211952995, con1482103504
     - serv1599256986
       - Inputs: con1653328292, con1849951292, con241744282
       - Outputs: con100012944, con1810216552, con406825148
     - serv1668689219
       - Inputs: con1653328292, con1849951292, con241744282
       - Outputs: con1257011377, con95711533
     - serv976005395
       - Inputs: con1348154594, con424848942, con588701442, con848610623
       - Outputs: con1189013645, con134421950, con1399563071, con30170533, con51881517, con633555781
     - serv283321609
       - Inputs: con10304228, con1189013645, con30170533, con53520061
       - Outputs: con1289781877, con1489681927, con149168694, con351525476, con730842958, con912923257
     - serv1738121452
       - Inputs: con1489681927, con149168694, con1631823443, con666530324, con912923257
       - Outputs: con1804686775, con1910780741, con556545125
     - serv1114869861
       - Inputs: con1489681927, con149168694, con1631823443, con666530324, con912923257
       - Outputs: con164119443, con189107477, con582761525
     - serv1876985918
       - Inputs: con2129932951, con582761525, con764841824
       - Outputs: con1498488754, con1869203452, con801503557, con851887673
     - serv1184302094
       - Inputs: con2049645207, con323056349, con761564774
       - Outputs: con1357575604, con365862042
     - serv491618308
       - Inputs: con2049645207, con323056349, con761564774
       - Outputs: con1335046394, con1772940636, con427511809
2. Web Service Challenge testset 2:
   - Inputs: con1498435960, con189054683, con608925131, con1518098260
   - Outputs: con357002459
   - Solution: sequence(splitJoin(sequence(serv1189164894, serv496481108), serv1258597127, serv2020713184), serv1328029360)
   - List of atomic processes:
     - serv1189164894
       - Inputs: con1233815228, con1498435960, con1518098260, con189054683
       - Outputs: con2040171441, con2050616774, con2085025818, con915123190
     - serv496481108
       - Inputs: con2050616774, con699658208, con915123190
       - Outputs: con115731217, con1545953204, con276510710, con29503594, con395302698
     - serv1258597127
       - Inputs: con1233815228, con1498435960, con1518098260, con189054683
       - Outputs: con1531820643, con1609445520, con965917446
     - serv2020713184
       - Inputs: con1233815228, con1498435960, con1518098260, con189054683
       - Outputs: con1027771256, con140513116, con2027267284, con2085025818, con674466131, con699658208, con763764707, con781788463, con794896663
     - serv1328029360
       - Inputs: con1038626729, con146453033, con395302698, con794896663, con798787896, con813330597, con918400240
       - Outputs: con368472115, con669754580, con841389546

Fig. 7 Description of the web services compositions 1 and 2 used for testing on repositories from *WSC 2008*

correct compositions have been obtained in many different ways. A valid composition was found in all the runs.

- *OWL-S TC V2.2*–5: this composition is an *split* of atomic processes. In all the runs, a valid composition was obtained. Moreover, the execution time was always the lower one (1). The number of atomic processes was also, most of the times, the minimum.
- *WSC 2008*–1: this composition is very complex, as it requires the *sequence* of 6 processes, three of them composite processes. These composite processes are constructed with *split* and *splitJoin* control structures (this is one of the possible solutions to this composition). In all the runs, a valid solution was found. Results show a very low variability in the execution time of the composite process. On average, the execution time (6) is the same of the solution shown in Fig. 7. The number of atomic processes is higher than expected (10), as other valid solutions have been found with more than 10 atomic services.
- *WSC 2008*–2: this is also a complex composition, with three nested control structures: a *sequence*, an *splitJoin*, and a *sequence*. The first *sequence* controls two services. The first of them is a composite service of type *splitJoin* of three services. Again, the first of them is a composite process of type *sequence* over two

1. Web Service Challenge testset 5:
   - Inputs: con428391640, con2100909192
   - Outputs: con1092196197, con1374634550, con2055848680
   - Solution: sequence(serv247333572, splitJoin(serv316765805, serv386198038, serv1840997881), serv1217746328, splitJoin(serv525062504, sequence(serv2049294580, serv663926970), serv40675417, serv802791436), splitJoin(serv110107650, serv872223669, serv248972116, serv1703771959), splitJoin(sequence(serv1011088135, serv318404349), serv1080520368), split(serv387836582, sequence(serv1842636425, serv457268815)))
   - List of atomic processes:
     - serv247333572
       - Inputs: con1368696763, con2100909192,
       - Outputs: con1060243885, con1837312783, con1899780776, con28797675,
     - serv316765805
       - Inputs: con1822359942, con384151484, con98229908,
       - Outputs: con1196037436, con1275915002, con2140027657, con507448888, con6676551, con81844658,
     - serv386198038
       - Inputs: con1822359942, con384151484, con98229908,
       - Outputs: con1681242749, con17328019, con2082065080, con960704019,
     - serv1840997881
       - Inputs: con1822359942, con384151484, con98229908,
       - Outputs: con1082569052, con220709124, con369404740,
     - serv1217746328
       - Inputs: con1196037436, con1606076734, con1749856794, con359573590, con369404740, con418968538,
       - Outputs: con1516982201, con1897732092, con361212134, con671505431, con82458841, con834129565,
     - serv525062504
       - Inputs: con131614591, con1602799684, con361212134, con844574898,
       - Outputs: con215587433, con254912033,
     - serv2049294580
       - Inputs: con131614591, con1602799684, con361212134, con844574898,
       - Outputs: con1121483512, con1464139261, con1548114195, con1944839158, con486968400,
     - serv663926970
       - Inputs: con1133159303, con1529884266, con2022463997,
       - Outputs: con1269975085, con1310117911, con417330032, con524244316, con578519665,
     - serv40675417
       - Inputs: con131614591, con1602799684, con361212134, con844574898,
       - Outputs: con1137664719, con1625739034, con1963069049, con459113456,
     - serv802791436
       - Inputs: con131614591, con1602799684, con361212134, con844574898,
       - Outputs: con1116567918, con1412526779, con32688908, con414052982, con540015383,
     - serv110107650
       - Inputs: con1302131402, con1582113061, con1739819509, con1834035733, con1963069049, con27159169, con459113456,
       - Outputs: con1120051103, con2077355659,
     - serv872223669
       - Inputs: con1302131402, con1582113061, con1739819509, con1834035733, con1963069049, con27159169, con459113456,
       - Outputs: con308165151, con374114199, con427981500,
     - serv248972116
       - Inputs: con1302131402, con1582113061, con1739819509, con1834035733, con1963069049, con27159169, con459113456,
       - Outputs: con1091786019, con1224710568, con1967574465, con292598089,
     - serv1703771959
       - Inputs: con1302131402, con1582113061, con1739819509, con1834035733, con1963069049, con27159169, con459113456,
       - Outputs: con1210374002, con1395731351, con1928864048, con85939934,
     - serv1011088135
       - Inputs: con1070281170, con1818468709, con1882167160, con1967574465, con241599790, con427981500, con84711568,
       - Outputs: con1620003160, con1753748027, con2101933515, con379850073, con76108784,
     - serv318404349
       - Inputs: con1141966130, con1753748027, con2101933515, con570327021,
       - Outputs: con1759687944, con841297848,
     - serv1080520368
       - Inputs: con1070281170, con1818468709, con1882167160, con1967574465, con241599790, con427981500, con84711568,
       - Outputs: con1119844968, con1264035168, con1613245017, con589171133,
     - serv387836582
       - Inputs: con1119844968, con1613245017, con1759687944, con658603366,
       - Outputs: con1324864617, con1374634550, con1876431248, con240985607, con374934517, con424090267, con529978098, con793166421, con832491021,
     - serv1842636425
       - Inputs: con1119844968, con1613245017, con1759687944, con658603366,
       - Outputs: con124240173, con1286564378, con1687592844, con495775189,
     - serv457268815
       - Inputs: con495775189, con952511413, con998186070,
       - Outputs: con1740843832, con396235323, con465871599, con64435085, con885742047,

Fig. 8 Description of the web service composition 5 used for testing on repositories from *WSC 2008*

atomic processes. The composition algorithm was able to find a valid solution in all the runs. Moreover, the execution time of the composite process was very close to the minimum one (3), and also the number of atomic processes was close to the minimum (5).

– *WSC 2008*–5: this is the most complex composition of all the tests and, also, the repository is the largest one (1,090 services). One of the solutions to this composition (the one described in Fig. 8) uses three different control structures (*sequence*, *split*, and *splitJoin*), some

Table 1 Average results ($\overline{\chi} \pm \sigma$) for the test examples

Example	Search time (ms)	$\frac{\left\|I^{n}_{root} \cap I_{obj}\right\|}{\left\|I_{obj}\right\|}$	$\frac{\sum_{i}^{\left\|o_{obj}\right\|} \frac{1}{DO_i+1}}{\left\|O_{obj}\right\|}$	Fitness	*runPath*	*#atomicProcess*
OWL-S TC V2.2–1	749.00 ± 364.10	1.00 ± 0.00	1.00 ± 0.00	1.0000 ± 0.0000	1.00 ± 0.00	1.00 ± 0.00
OWL-S TC V2.2–2	484.50 ± 139.20	1.00 ± 0.00	1.00 ± 0.00	0.9750 ± 0.0000	2.00 ± 0.00	2.00 ± 0.00
OWL-S TC V2.2–3	473.60 ± 76.19	1.00 ± 0.00	1.00 ± 0.00	0.9750 ± 0.0000	2.00 ± 0.00	2.00 ± 0.00
OWL-S TC V2.2–4	3010.20 ± 422.91	1.00 ± 0.00	1.00 ± 0.00	0.9296 ± 0.0042	2.20 ± 0.40	5.70 ± 1.19
OWL-S TC V2.2–5	1098.30 ± 240.72	1.00 ± 0.00	1.00 ± 0.00	0.9654 ± 0.0019	1.00 ± 0.00	3.30 ± 0.46
WSC 2008–1	6919.70 ± 1612.99	1.00 ± 0.00	1.00 ± 0.00	0.9112 ± 0.0012	6.00 ± 1.26	15.8 ± 5.71
WSC 2008–2	11137.30 ± 3106.75	1.00 ± 0.00	1.00 ± 0.00	0.9233 ± 0.0023	3.50 ± 0.67	6.00 ± 0.89
WSC 2008–5	95390.20 ± 43521.30	1.00 ± 0.00	1.00 ± 0.00	0.9069 ± 0.0011	9.20 ± 2.96	49.90 ± 16.84

of them nested three times. There are a total of 9 control structures and 20 atomic processes. The solution is a *sequence* of seven processes. Five of them are composite processes of type *split* and *splitJoin*. Moreover, some of these processes have other composite processes nested. For example, the second *splitJoin* has four processes, and the second one is a *sequence* of two atomic processes. Of course, this composition cannot be obtained with sequence-based compositions. Although the complexity of the solution, the proposed composition algorithm was always able to find a valid solution. Moreover, the execution time was very low. On the other hand, the number of atomic processes was, on average, over the minimum one.

In summary, the performance of the algorithm is very good. The tests have been selected to cover different types of compositions, using several control structures. Moreover, the complexity of the tests is really high (three of them come from the *WSC 2008*), with up to nine control structures in a composition, control structures nested up to three times, and more than twenty atomic services in some of the obtained solutions. Although these complex tests, the composition algorithm was always able to find a valid solution: in the 80 runs the results were always valid. Also, the execution time of the obtained solutions (*runPath*) was the lowest (or close to the minimum), which reflects the ability of the algorithm to exploit the different control structures that it can manage. Finally, the number of atomic processes of the solutions was, in most of the tests, close to the minimum.

6 Conclusions

A genetic programming algorithm for web services composition has been presented. The algorithm is able to compound services using different control structures, generates compositions following a context-free grammar, and manages explicitly the attributes updating. A full validation has been done for eight different composition problems coming from four different repositories (three of them from the Web Service Challenge 2008) with 158, 558, 1,000, and 1,090 services, showing a very good performance. In all the tests and runs a valid solution was found, indicating that the algorithm is robust and reliable for different repositories. Moreover, the execution times of the obtained composite processes were also low, showing the ability of the algorithm to exploit the available control structures. Also, the search times of the evolutionary algorithm are quite low, allowing to use our proposal on-line.

Acknowledgments This work was supported by the Spanish Ministry of Science and Innovation under Grant TSI2007-65677-C02-02. Manuel Mucientes is supported by the *Ramón y Cajal* program of the Spanish Ministry of Science and Innovation.

References

1. Aiello M, van Benthem N, Khoury E (2008) Visualizing compositions of services from large repositories. In: Proceedings of the fifth IEEE conference on enterprise computing, e-commerce and e-services (EEE'08). IEEE Computer Society, Washington, DC, pp 359–362.
2. Alamri A, Eid M, Saddik AE (2006) Classification of the state of the art dynamic web services composition techniques. Int J Web Grid Serv 2(2):148–166
3. Alonso G, Casati F, Kuno H, Machiraju V (2003) Web services. Springer, Berlin
4. Anderson BB, Hansen JV, Lowry PB (2009) Creating automated plans for semantic web applications through planning as model checking. Expert Syst Appl 36(7):10595–10603
5. Aversano L, di Penta M, Taneja K (2006) A genetic programming approach to support the design of service compositions. Int J Comput Syst Sci Eng 4:247–254
6. Bansal A, Blake MB, Kona S, Bleul S, Weise T, Jaeger MC (2008) WSC-08: continuing the web services challenge. In: Proceedings of the 5th IEEE international conference on

enterprise computing, e-commerce and e-services (EEE'08). IEEE, pp 351–354

7. Bertoli P, Pistore M, Traverso P (2010) Automated composition of web services via planning in asynchronous domains. Artif Intell 174(3–4):316–361
8. Blum AL, Furst ML (1997) Fast planning through planning graph analysis. Artif Intell 90(1–2):281–300
9. Chang W-C, Wu C-S, Chang C (2005) Optimizing dynamic web service component composition by using evolutionary algorithms. In: Proceedings of the 2005 IEEE/WIC/ACM international conference on web intelligence (WI'05). IEEE Computer Society, Washington, DC, pp 708–711
10. Curbera F, Goland Y, Klein J, Leymann F, Roller D, Thatte S, Weerawarana S (2002) Business process execution language for web services, version 1.0, November 2002. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation
11. Curbera Francisco, Nagy William A., Weerawana Sanjiva (2001) Web Service: Why and How. In: Proceedings of the OOPSLA-2001 workshop on object-oriented services. Tampa, Florida, USA
12. Dustdar S, Schreiner W (2005) A survey on web services composition. Int J Web Grid Serv 1(1):1–30
13. Hoffmann Jörg, BP, Pistore M (2007) Web service composition as planning, revisited: in between background theories and initial state uncertainty. In: Proceedings of the 22nd national conference on artificial intelligence (AAAI'07). AAAI Press, pp 1013–1018
14. Klusch M, Gerber A (2006) Fast composition planning of owl-s services and application. In: Proceedings of the European conference on web services (ECOWS'06). IEEE Computer Society, Washington, DC, pp 181–190
15. Kuster U, Konig-Ries B, Krug A (2008) Opossum—an online portal to collect and share sws descriptions. In: Proceedings of the 2th IEEE international conference on semantic computing (ICSC 2008). IEEE Computer Society, Santa Clara, pp 480–481
16. Madhusudan T, Uttamsingh N (2006) A declarative approach to composing web services in dynamic environments. Decis Support Syst 41(2):325–357
17. Martin D, Burstein M, Hobbs J, Lassila O, McDermott D, McIlraith S, Narayanan S, Paolucci M, Parsia B, Payne T, Sirin E, Srinivasan N, Sycara K (2004) OWL-S: semantic markup for web services. World Wide Web Consortium (W3C), November 2004. Available at http://www.w3.org/Submission/OWL-S/
18. Nau D, Au T-C, Ilghami O, Kuter U, Murdock W, Wu D, Yaman F (2003) SHOP2: an HTN planning system. J Artif Intell Res 20(4):379–404
19. Oh S-C, Lee D, Kumara SRT (2006) A comparative illustration of ai planning-based web services composition. SIGecom Exch 5(5):1–10
20. Oh S-C, Lee D, Kumara SRT (2008) Effective web service composition in diverse and large-scale service networks. IEEE Trans Serv Comput 1(1):15–32
21. Pistore M, Marconi A, Bertoli P, Traverso P (2005) Automated composition of web services by planning at the knowledge level. In Proceedings of the 19th international joint conference on artificial intelligence (IJCAI'05). Morgan Kaufmann Publishers Inc, San Francisco, pp 1252–1259
22. Rao J, Küngas P, Matskin M (2006) Composition of semantic web services using linear logic theorem proving. Inform Syst 31(4):340–360
23. Ren K, Liu X, Chen J, Xiao N, Song J, Zhang W (2008). A QSQL-based efficient planning algorithm for fully-automated service composition in dynamic service environments. In: Proceedings of the 2008 IEEE international conference on services computing (SCC'08). IEEE Computer Society, Washington, DC, pp 301–308
24. Russell SJ, Norvig P (2009) Artificial intelligence: a modern approach. Prentice Hall, Englewood Cliffs
25. Sirin E, Parsia B, Wu D, Hendler J, Nau D (2004) Htn planning for web service composition using shop2. J Web Semant 1(4):377–396
26. Vanrompay Y, Rigole P, Berbers Y (2008) Genetic algorithm-based optimization of service composition and deployment. In: Proceedings of the 3rd international workshop on services integration in pervasive environments (SIPE'08). ACM, New York, pp 13–18
27. Wu Z, Gomadam K, Ranabahu A, Sheth AP, Miller JA (2007) Automatic composition of semantic web services using process and data mediation. In: Proceedings of the 9th international conference on enterprise information systems (ICEIS'07). Funchal, Portugal, pp 453–461
28. Xiangbing Z (2010) Semantics web service characteristic composition approach based on particle swarm optimization volume 56 of Lecture Notes in Electrical Engineering. Springer, pp 279–287
29. Zheng X, Yan Y (2008) An efficient syntactic web service composition algorithm based on the planning graph model. In: Proceedings of the 2008 IEEE international conference on web services (ICWS'08). IEEE Computer Society, Washington, DC, pp 691–699